USC Viterbi
School of Engineering

Ming Hsieh Department of Electrical Engineering

EE 109L - Introduction to Embedded Systems

Spring 2019 Project Assignment
Rangefinder

1 Introduction

This semester's class project is to build a device that measures the range to an object by using an ultrasonic rangefinder and displays the range on an LCD and also relays that distance to a remote device. The rangefinder will have the following features.

- An ultrasonic rangefinder for measuring distances up to 400cm.
- Buttons to initiate a range measurement and set distance thresholds.
- An LCD display for showing the measured range. The display is also used for setting the distance thresholds and displaying the distance received from the remote device.
- A control knob for setting the local and remote distance thresholds.
- Two LEDs for showing whether the measured distance is less than or greater than a threshold.
- A buzzer for playing an alarm tone.
- A serial interface to another rangefinder unit. When the local unit measures a distance it sends it to the remote unit. When a distance is received from the remote unit it is displayed on the LCD.

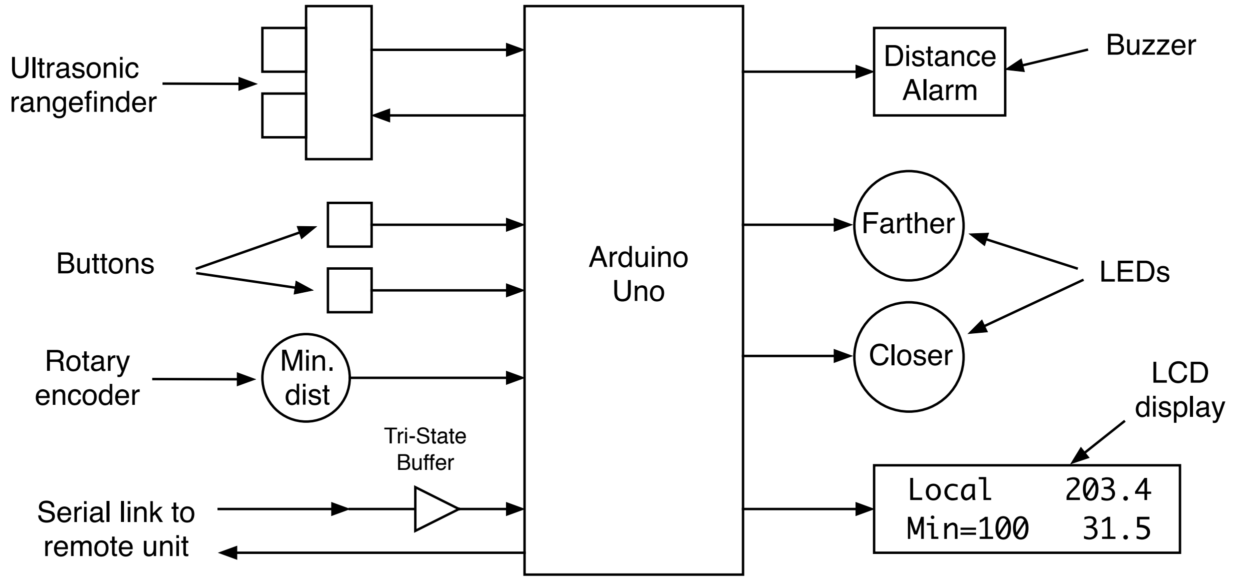


Figure 1: Block diagram of the rangefinder device

2 Operation of the Rangefinder

The operation of the rangefinder is relatively simple.

- Each time the "Acquire" button is pressed a signal is sent to the ultrasonic sensor to initiate a range measurement by sending out an ultrasonic pulse.
- As the sensor makes a range measurement, it sends to the Arduino a $0 \rightarrow 1 \rightarrow 0$ pulse that varies in width proportional to the range to the object that reflected the ultrasonic pulse. The Arduino monitors the received pulse watching for the initial rising edge and the falling edge at the end of the pulse to determine the length of the pulse. Once this is known it performs the calculations to find the corresponding distance.
- Whenever a range measuring event has completed, the unit displays the distance **in centimeters** with one decimal place (e.g. "27.6") on the LCD display.
- When a range measurement is completed, the local unit transmits the distance over the serial port to the remote unit. The protocol used for sending and receiving the range data is specified in Section 4.6. When a rangefinder receives distance data from another unit, it displays the received distance in centimeters on its LCD. For this project it should be possible to do a "loop back" of the transmitted data to the received data port so the rangefinder system is essentially sending the data to itself.
- An "Adjust" button is used to determine which of two range thresholds are adjusted by the rotary encoder, either a local threshold that is compared to the range measured by the local unit, or a remote range that is compared to the distance that has been received from another unit. Each time the button is pressed the mode toggles from one mode to the other and the mode the unit is currently in is displayed on the LCD as either "Local" or "Remote".
- If the user turns the rotary encoder knob, the selected range threshold number is changed up or down and displayed on the LCD. This number is an integer value and can change from 1 to 400 cm.
- Whenever either range threshold is changed, the new value is stored in the Arduino's EEPROM non-volatile memory so the values are retained even if the power is turned off. Whenever the rangefinder is turned on or restarted, the threshold values are read from the EEPROM memory making it unnecessary for the user to set the distance thresholds each time the device is turned on.
- The two LEDs are used to show how the local range measurement compares to the local range threshold that was set by the rotary encoder. If the range measurement made by the local device is greater than the local range threshold, the green LED comes on. If the local range is less than the threshold, the red LED comes on.
- The buzzer is used to indicate when the remote device measured a distance less than the remote range threshold set on the local device. If a distance value received from the other rangefinder is less than the remote threshold distance the unit sounds a short alarm using the buzzer.

3 Hardware

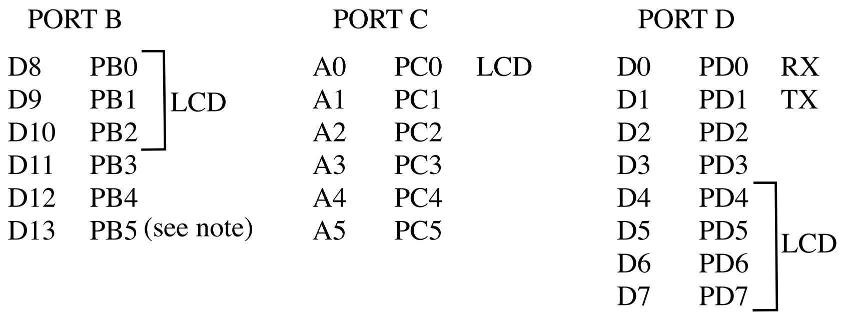
Most of the components used in this project have been used in previous labs, and your C code from the other labs can be reused if that helps. The buzzer was used in Lab 6 (ADC) and you can use some of that code to play a tone on the buzzer. The rotary encoder is the same as in Lab 7. You should use the encoder code that uses interrupts rather than polling. The serial link is similar (but not the same) as the serial link used in Lab 9

The buttons, LEDs, rotary encoder, buzzer and 74LS125, and any required resistors should all be mounted on your breadboard. It's strongly recommended that you try to wire them in a clean and orderly fashion. Don't use long wires that loop all over the place to make connections. You will have about 14 wires going from the Arduino to the breadboard so don't make matters worse by having a rat's nest of other wires running around on the breadboard. Feel free to cut off the leads of the LEDs and resistors so they fit down close to the board when installed.

Make use of the bus strips along each side of the breadboard for your ground and $+5V$ connections. Use the red for power, blue for ground. There should only be one wire for ground and one for $+5V$ coming from your Arduino to the breadboard. All the components that need ground and/or $+5V$ connections on the breadboard should make connections to the bus strips, not wired back to the Arduino.

3.1 Arduino Ports

The Arduino Uno has 20 general purpose I/O lines in Ports B, C and D. However most of these are shared with other modules and can not be used for general purpose I/O if that module has to be used. For example, in this project you will be using the serial communications modules which requires using specific I/O lines. In addition, the LCD shield requires using certain port bits. Fig. 2 shows which I/O port bits are allocated for various purposes.



PORT B			PORT C			PORT D		
D8	PB0	LCD	A0	PC0	LCD	D0	PD0	RX
D9	PB1	LCD	A1	PC1		D1	PD1	TX
D10	PB2	LCD	A2	PC2		D2	PD2	
D11	PB3		A3	PC3		D3	PD3	
D12	PB4		A4	PC4		D4	PD4	LCD
D13	PB5	(see note)	A5	PC5		D5	PD5	LCD
						D6	PD6	LCD
						D7	PD7	LCD

Figure 2: Use of I/O port bits in this project

LCD - The LCD shield uses PD4-PD7 for the data lines, PB0 and PB1 for control, PB2 for the backlight, and PC0 is the analog signal from the buttons.

Serial Data - The USART0 serial interface module uses PD0 and PD1 for received (RX) and transmitted (TX) data, respectively.

Port B, bit 5 - This I/O bit has an LED connected to it on the Arduino board and this can prevent it from being used as an input. You might be able to use it as an output, but avoid using it as an input. In addition, **do not use it to control the tri-state buffer**. While it can be used for some other output signal, the LED causes it to not work correctly during the flash programming to control the buffer.

When working on your design to read input signals and produce output signals make sure to use only the I/O port bits that are not already in use by one of the above modules or the shield. Also think about how your program will be communicating with the devices and how that might affect your decision as to which ports to use for the various devices. For example, if you want to use Pin Change Interrupts to monitor an input from the range sensor, it might be a good idea to **not** have that input on the port that the rotary encoder is using since your code for that may use Pin Change Interrupts and you may want to have the encoder handled by a different ISR than the sensors. A little planning in advance as to how your program will work can lead to significant simplifications in the software.

3.2 Ultrasonic Range Sensor

The range sensor is a single module that can be mounted on your breadboard. It requires $+5V$ power and ground, and two digital signals. One signal is an output from the Arduino and is used to initiate a range measurement. The other signal is an input to the Arduino and is the pulse generated by the sensor to indicate the range.

Two pages from the manufacturer's datasheet for the sensor are attached at the end of this document.

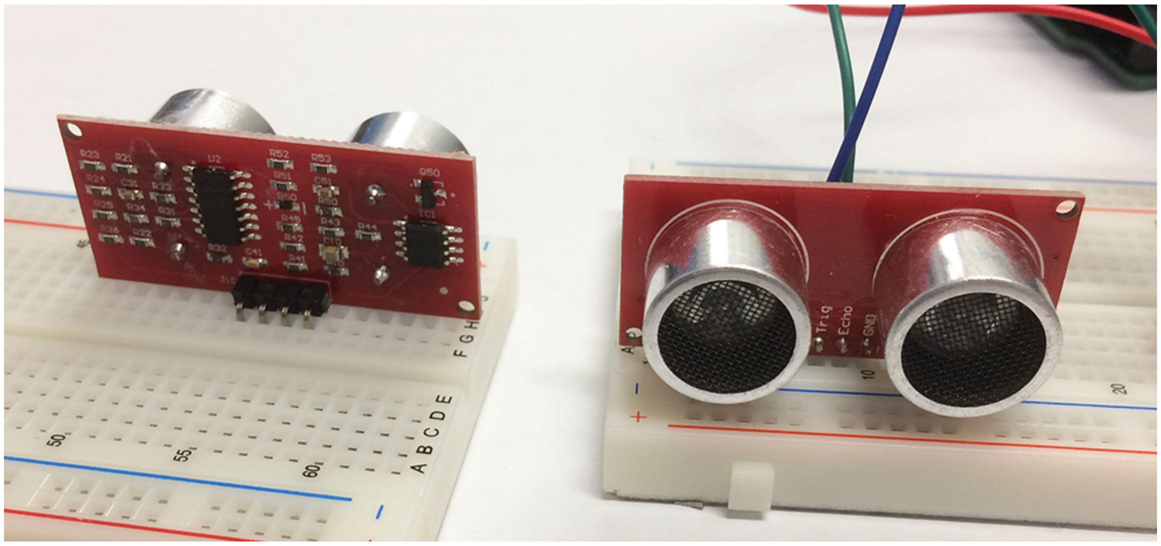


Figure 3: Ultrasonic range sensors

The four pins at the bottom of the sensor should be installed in four separate blocks of holes on the breadboard. The purpose of each pin is shown on the front of the sensor: `VCC`, `Trig`, `Echo`, `GND`. The trigger (output) and echo (input) signals are digital signals so those can be interfaced to the digital I/O port bits of the Arduino. Read the datasheet for information on the format of these signals.

3.3 Serial Interface

The serial interface between rangefinder devices will use an serial data link to send the distance data between the units. The serial input and output of the Arduino uses voltages in the range of 0 to +5 volts. These are usually called "TTL compatible" signal levels since this range was standardized in the transistor-transistor logic (TTL) family of integrated circuits. Normally for RS-232 communication these voltages must be converted to positive and negative RS-232 voltages levels in order for it to be compatible with another RS-232 device. However for this project we will skip using the voltage converters and simply connect the TTL level signals between the project boards.

For all the rangefinder devices to be capable of communicating with others, use the following parameters for the USART0 module in the Arduino. Refer to the slides from the lecture on serial interfaces for information on how to configure the USART0 module for these settings.

- Baud rate = 9600
- Asynchronous operation
- Eight data bits
- No parity
- One stop bit

3.4 Tri-State Buffer

As was seen in Lab 9, if the received data from another device is connected directly to the RX input (Arduino port D0) it creates a conflict with a signal used to program the Arduino's microcontroller. Both the other device and the programming hardware try to put an active logic level on the D0 input and this can prevent the programming from succeeding. When this happens you will get an error messages like this.

```
avrdude: stk500_recv(): programmer is not responding
avrdude: stk500_getsync() attempt 1 of 10: not in sync: resp=0x00
avrdude: stk500_recv(): programmer is not responding
avrdude: stk500_getsync() attempt 2 of 10: not in sync: resp=0x00
```

The solution for this is to use a tri-state gate to isolate the other device from the D0 input until after the programming is finished. The gate you will using is a 74LS125 that contains four non-inverting buffers (see Fig. 4). Each of the four buffers have one input, one output and an enable input that controls the buffer's tri-state output. When the gate is enabled it simply copies the input logic level to the output ($0 \rightarrow 0, 1 \rightarrow 1$). However when the gate is disabled its output is in the tri-state or hi-Z state regardless of the input. In that condition, the gate output has no effect on any other signal also attached to the output.

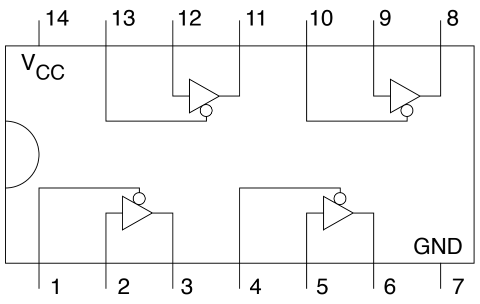


Figure 4: 74LS125 Tri-State buffer

As shown in Fig. 5, the received data from the other device should go to one of the buffer inputs, and the output of the buffer should be connected to the D0 port of the Arduino. The enable signal is connected to any I/O port bit that is available for use. When the Arduino is being programmed all the I/O lines become inputs and this will effectively put a logic one on the tri-state buffer's enable line. This disables the output (puts it in the hi-Z state) and the programming can take place. Once your program starts, all you have to do is make that I/O line an output and put a zero in the PORT bit. This will enable the buffer and now the other devices's received data will pass through the buffer to the RX serial port.

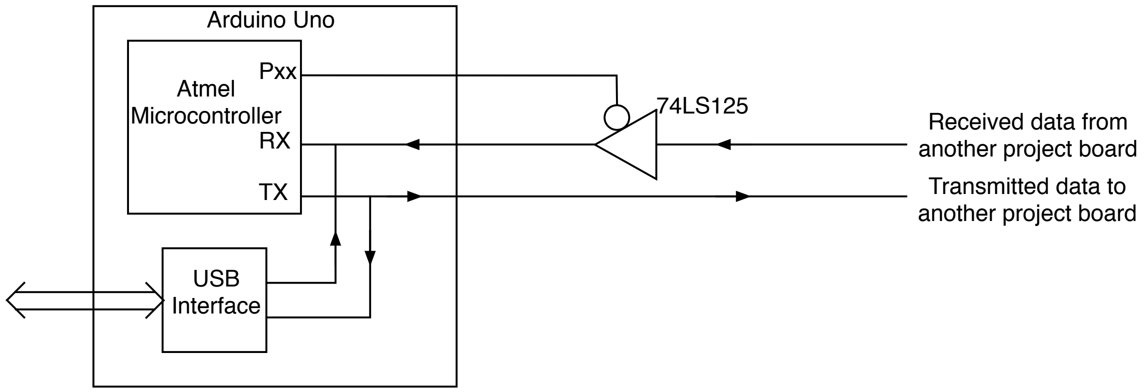

Arduino Uno
Atmel Microcontroller
Pxx
RX
TX
USB Interface
74LS125
Received data from another project board
Transmitted data to another project board

Figure 5: Using the 74LS125 Tri-State buffer to interface to another device

4 Software

Your software should be designed in a way that makes testing the components of the project easy and efficient. In previous labs we worked on putting all the LCD routines in a separate file and this practice should be continued here. Consider having a separate file for the encoder routines and its ISR, and another one for the serial interface routines. Code to handle the sensor can either be in a separate file or in the main program since there isn't much code for these. All separate code files must be listed on the `OBJECTS` line of the `Makefile` to make sure everything gets linked together properly.

4.1 Improving Your Makefile

In class we discussed how the "`make`" program uses the data in the "`Makefile`" to compile the various modules that make up a program. This project may require several source code files, some with accompanying "`.h`" header files, so the generic `Makefile` should be modified to describe this. For example, let's say you have four C files for the project and four header files:

- The main program is in `rangefinder.c` and has some global variables and functions declared in `rangefinder.h`
- The LCD routines are in `lcd.c` with global declarations in `lcd.h`
- The functions to handle the rotary encoder are in `encoder.c` with global declarations in `encoder.h`
- The functions for the serial I/O are in `serial.c` with global declarations in `serial.h`

Let's also say that `rangefinder.h` is "included" in all the C files, and the header files for the LCD, encoder and serial routines are included in the `rangefinder.c` file. In this situation, the following lines should be added to the `Makefile` after the "`all: main.hex`" and before the "`.c.o`" line as shown below.

```
all:    main.hex

rangefinder.o: rangefinder.c rangefinder.h lcd.h encoder.h serial.h
lcd.o:         lcd.c lcd.h rangefinder.h
encoder.o:     encoder.c encoder.h rangefinder.h
serial.o:      serlal.c serial.h rangefinder.h

.c.o
```

Adding all the dependencies to the Makefile will make sure that any time a file is edited, all the affected files will be recompiled the next time you type `make`.

4.2 Determining the Sensor Pulse Width

The pulse that comes back from the sensor can be connected to a I/O port bit that has the Pin Change Interrupt enabled. The ISR will be invoked once on the rising edge of the pulse (start of the measurement) and again on the falling edge (end of the measurement).

The ISR can control one of the timers to produce a count of how many counter clock cycles it took to get from the rising edge to the falling edge. For determining the width of the measurement pulse from the sensor, it is recommended that you use the 16-bit TIMER1 to count at a known frequency. At the start of the program the timer can be configured but left in the stopped state (all prescaler bit cleared to zeros). When the ISR is invoked by the start of the pulse, the timer's count can be set to zero with the statement

```
TCNT1 = 0;
```

and then the timer is started by loading the correct prescalar bits.

When the ISR is invoked again by the end of the pulse, the timer can be stopped by setting the prescalar bits to all zeros and the value in the count registered can be examined.

```
pulse_count = TCNT1;
```

Since the rate at which the timer was counting is known, the count value can be used to determine how long the pulse was in the high state.

4.3 Time-out Function on Pulse Width Measurement

We want our rangefinder to be robust and not subject to problems that may be caused by electrical noise spikes on the signal lines. For example, a noise spike on the sensor output may be incorrectly taken to be the start of a measurement pulse, but the Arduino may then sit there forever waiting for the falling edge of the pulse. To prevent this from happening we want to install a "watch dog timer" function that will prevent the rangefinder from getting locked up.

The manufacturer's datasheet tells what the maximum range is for the sensor. From that we can calculate what the highest count value is that we would ever see from our TIMER1 if it's operating properly. If the TIMER1 count ever goes above this value, we can assume that something has gone wrong and we want to abandon this measurement. You should set up your TIMER1 so that it generates an interrupt if this maximum count value is reached. The ISR can then stop the timer and reset whatever is needed to discard this measurement and wait for a new measurement to be initiated.

4.4 Buzzer

In Lab 7 you worked with producing tones of different frequencies from the buzzer. Those tones were done with code that used delays of half the desired output period between operations to make the output signal go high or low. The result was a squarewave signal at the desired frequency.

The problem with this method is that the program is locked into the delay routines while they measure out the selected delay time. A better way to create the tones is by using a timer to generate interrupts at the desired rate, and when each interrupt occurs the ISR changes the state of the output bit driving the buzzer.

To receive full credit for the buzzer output generation, use one of the two 8-bit timers (TIMER0 or TIMER2) to generate the buzzer signal. If you use the delay routines as in Lab 7, you will still receive partial credit for this task.

4.5 EEPROM Routines

The avr-gcc software includes several routines that you can use for accessing the EEPROM. To use these functions, your program must have this "`include`" statement at the beginning of the file that uses the routines.

```
#include <avr/eeprom.h>
```

eeprom_read_word - This function reads two bytes from the EEPROM starting at the address specified and returns the 16-bit value. It takes one argument, the EEPROM address (0-1023) to read from. For example to read a word from EEPROM address 100:

```
x = eeprom_read_word((void *) 100);
```

eeprom_update_word - This function writes two bytes to the EEPROM starting at the address specified. It takes two arguments, the address to write to and the 16-bit value of the word to be stored there. For example to write the word 0x2f47 to address 200 in the EEPROM:

```
eeprom_update_word((void *) 200, 0x2f47);
```

Your code should use the above routines to store the minimum distance values in the EEPROM whenever it has been changed. Since the minimum distance is from 1 to 400, this only requires writing a single word to the EEPROM for each value. You can choose any addresses in the EEPROM address range (0 to 1023) to store the values. When your program starts up it should read the values from the EEPROM, but it must then test the values to see if they are valid. If the EEPROM has never been programmed, it contains all

0xFF values. If you read the EEPROM data and one of the values is not in the range 1 to 400, then your program should ignore this number and revert to using a default minimum distance value that is defined in your source code.

Warning! The EEPROM on the microcontroller can be written to about 100,000 times and after that it will probably stop working. This limit should be well beyond anything we need for this project but it's very important that you make sure you don't have the above EEPROM writing routines in some sort of loop that might go out of control and do 100,000 writes before you realize the program isn't working right. **Your code should only write to the EEPROM when the minimum distance value has changed**.

4.6 Serial Interface Routines

The serial data link between two rangefinders uses a simple protocol:

- Data sent using ASCII characters for easier debugging
- Accommodates varying sizes of distance data text strings
- Allows the system to recover from errors such as a partially transmitted or garbled data packet

In many devices that use serial links these features are implemented using relatively complex data structures and interrupt service routines so the processor does not have to spend much time doing polling of the receiver and transmitter. We'll do it in a simpler manner that should work fine for this application.

The protocol for communicating the measured distance value between two rangefinder units will consist of a string of bytes in this format:

- The start of the string is indicated by the `'@'` character.
- Up to four ASCII digits ('0' through '9') representing the distance in **millimeters per second** as an integer value (no fractional distance). For example if the measured distance was 23.7 cm, it should send the three ASCII characters `237`. The maximum value that can be sent is determined by the maximum range of the sensor. You device should only send the necessary digits. For example if the distance is 7.4 cm, you only need to send `74`. You should not send `0074`.
- After all characters for the distance has been sent the end of the distance data string is indicated by sending the `'$'` character.

4.6.1 Transmitting Data

When your software determines that the distance measurement has been completed, it should call a routine that sends the characters for the distance to the remote unit. The serial link is much slower than the processor so the program has to poll the transmitter to see when it can put the next character to be sent in the transmitter's data register for sending. The `UCSR0A` register contains a bit (`UDRE0` - USART Data Register Empty) that tells when it's ready to be given the next character to send. While this bit is a zero, the transmitter is busy sending a previous character and the program must wait for it to become a one. Once the `UDRE0` bit becomes a one, the program can store the next character to be sent in the `UDR0` register.

```
while ((UCSR0A & (1 << UDRE0)) == 0) { }
UDR0 = next_character;
```

While your program is waiting for all the characters to be transmitted it should still respond to interrupts from modules with interrupts enabled, but it does not have to reflect any changes on the display until after all the data has been sent and it's back in the main program loop.

4.6.2 Receiving Data

Receiving the distance data from the remote unit is a bit more complicated since you have no idea when the remote unit will send the data. One simple way to implement this is to have your program check for a received character each time through the main loop. If one has been received, then call a function that waits for the rest of the characters and when complete displays the distance on the LCD. Unfortunately this method of receiving characters has one very bad design flaw in it. If for some reason the string is incomplete, maybe only the first half of the string was sent, the device will sit in the receiver subroutine forever waiting for the rest of the data and the `'$'` that marks the end of the transmission.

A better solution, and one that should be implemented in your program, is to use interrupts for the received data. Receiver interrupts are enabled by setting the `RXCIE0` bit to a one in the `UCSR0B` register. When a character is received the hardware executes the ISR with the vector name "`USART_RX_vect`".

For reading the incoming distance data, each time a character is received, an interrupt is generated and the ISR determines what to do with the character. After all the characters have been received, the ISR sets a global variable to indicate that a complete remote distance value has been received and is available. When the main part of the program sees this variable has been set, it gets the value and displays it. By using the interrupts, the program is never stuck waiting for a character that might never come.

It is also important to consider **all** the possible errors that might occur in the transmission of the date, such as missing start (`'@'`) character, missing or corrupted distance characters, missing end (`'$'`) character, etc. The software must make sure all of these situations are handled cleanly and don't leave the device in an inoperable state.

To implement this, use the following variables.

- A 5 byte buffer for storing the data from the remote sensor as it comes in (the 4 data bytes and a `'\0'` byte, at the end.)
- A global variable to act as a data started flag that tells whether or not the start character (`'@'`) has been received indicating data is to follow.
- A variable that tells how many data characters have been received and been stored in the buffer so far. This also tells the ISR where in the buffer it should store the next character.
- A global variable to act as a data valid flag to indicate that the `'$'` has been received and the buffer contains a valid distance string. This variable should be zero while receiving the distance data, and set to one only after receiving the `'$'` that marks the end of the sequence.

The ISR uses these three variables to properly receive the data.

- If the ISR receives a `'@'`, this indicates the start of a new distance data sequence **even if the previous sequence was incomplete**. Set the data start variable to a one, and clear buffer count to 0. Also set the the valid data flag to zero to indicate that you now have incomplete data in the buffer.
- If the ISR receives a `'$'`, and the buffer count is greater than zero (meaning the sequence has started) set the valid data flag variable to a one to indicate complete data in the buffer. However if the end transmission character is received but there is no distance data (nothing in the buffer between the `'@'` and the `'$'`, the flag variable should not be set to a one.
- If a sequence has started and a character in the range of 0 to 9 is received, store it in the next buffer position and increment the buffer count. If after the start of a sequence something other than the number 0 through 9 or the end of transmission marker `'$'` is received, reset the data started flag to zero to discard what has been received so far. This will set up the ISR to wait for the next transmission. Your code should also make sure there is room in the buffer for the data. If the data tries to overrun the length of the buffer this would imply two transmissions have somehow been corrupted into looking like one, and in this case you should set the data started flag back to zero to discard this transmission.

The main program can check the data valid variable each time through the main loop. When it sees it has been set to a one, it can call a function to convert the distance data from from a string of ASCII characters to a fixed-point binary number (see Sec 4.7). It should probably also clear the data valid variable to a zero so it doesn't re-read the same data the next time through the loop.

4.7 Using `sscanf` to Convert Numbers

In Lab 5 you learned how to use the "`snprintf`" function to convert a binary number into a string of ASCII characters. Now we need to go the other way, from a string of ASCII characaters into single binary fixed-point number. For this we can use the "`sscanf`" function that is part of the the standard C library.

Important: As with using `snprintf`, in order to use `sscanf` you must have the following line at the top of the program with the other `#include` statements.

```
#include <stdio.h>
```

The `sscanf` function is called in the following manner

```
sscanf(buffer, format, arg1, arg2, arg3, ...);
```

where the arguments are

buffer – A `char` array containing the items to be converted to binary values.

format – The heart of the `sscanf` function is a character string containing formatting codes that tell the function exactly how you want it to convert the characters it finds in input string. More on this below.

arguments – After the `format` argument comes zero or more **pointers** to where the converted values are to be stored. For every formatting code that appears in the `format` argument, there must be a corresponding argument containing the a pointer to where to store the converted value.

The `format` argument tells `sscanf` how to format the output string and has a vast number of different formatting codes that can be used. The codes all start with a percent sign and for now we will only be working with one of them:

%d – Used to format decimal integer numbers. When this appears in the format string, the characters in the input string will be interpreted as representing a decimal integer number, and they will be converted to the corresponding binary value. The result will be stored in the variable that the corresponding argument points to.

The `format` string must have the same number of formatting codes as there are arguments that follow it in the function call. Each formatting code tells how to convert something in the input string into its corresponding argument. The first code tells how to convert something that is stored where "`arg1`" points, the second code is for "`arg2`", etc.

Example: Assume you have a `char` array containing the characters representing three numbers. The code below would convert them into the three `int` variables.

```
char buf[] = "12 543 865";
int num1, num2, num3;

sscanf(buf, "%d %d %d", &num1, &num2, &num3);
```

The arguments are pointers to where the three values are to be stored by using the form "`&num1`" which makes the argument a pointer to `num1` rather than the value of `num1`. After the function has executed, the variables "`num1`", "`num2`" and "`num3`" will contain the binary values 12, 543 and 865.

Important: The "`%d`" formatting code tells `sscanf` that the corresponding argument is a pointer to an **`int`** (4 byte) variable. When it converts the characters to a binary value it will store it in 4 bytes. If you wish to store a value in a "`short`" (two bytes), or a "`char`" (one byte) variable, you **must** modify the format code. The formatting code "`%hd`" tells it to store a 2 byte `short`, and "`%hhd`" tells it to store a 1 byte `char`.

Here's the above example but modified to use three different variable types.

```
char buf[] = "12 543 865";
char num1;
short num2;
int num3;

sscanf(buf, "%hhd %hd %d", &num1, &num2, &num3);
```

5 Building Your Project

It's important that you test the hardware and software components individually before expecting them to all work together. Here's a one possible plan for putting it together and testing it.

1. Install the "Acquire" button on the board and add code to detect the "Acquire" button pressed. When the Acquire button is pressed generate a pulse on the port bit that will be connected to the `Trig` signal to the sensor. Check with the scope that the pulse is generated and is of the correct length.
2. Add the range sensor to the boards and connect the `Trig` and `Echo` lines to two of the Arduino digital I/O port bits. Put two channels of the scope on the `Trig` and `Echo` lines and observe what happens when the Acquire button is pressed. The sensor should produce a pulse in response to the trigger signal. Try holding your hand in front of the sensor and see if you can make the echo pulse width change by moving your hand farther and closer to the sensor as you trigger it.
3. Determine how to configure TIMER1 to use it to measure the width of the pulse. You need to pick a prescaler value that will make it run as fast as possible so as get the most accurate timing of the pulse width. However it must run at a speed where the count will still be less than the maximum 16-bit value (65,535) if the sensor returns the maximum width output pulse.
4. Write code to implement the Pin Change Interrupt ISR for the sensor output signal. This ISR needs to do different things depending on whether it's detecting the start of the pulse (zero timer count, start timer) or the end of the pulse (stop timer, set flag that measurement complete). For debugging, have the count value printed on the LCD and see if it changes as you make measurements at differing distances.
5. Convert the count value from the timer to the distance in **millimeters** that you will need later to send to the remote device, and then to distance in centimeters for displaying on the LCD. **This should be done without using any floating point arithmetic.** Write the distance value to the LCD after each range measurement is completed.
6. Check that your measurement calculation correctly handles distance greater than the specified maximum range by indicating on the LCD that the distance is too far.
7. Add the "Adjust" button to the the board and add code to detect when it has been pressed. Each time it's pressed it toggles the adjustment mode between local and remote, and this should be indicated on the LCD.
8. Install the rotary encoder on the board and add code to use the rotary encoder to set the range threshold values. It should adjust the threshold that has been selected by the "Adjust" button and show the value on the LCD. Check that this allows you to adjust boths values between 1 and 400 cm.
9. Write code to store the minimum distance values in the EEPROM, and read the EEPROM values when the program starts. Confirm that this is working by adjusting the distance values and cycling the power on the project. It should start up and display the distances you had set before. Make sure to add code that checks that the distances you loaded from the EEPROM are valid values.
10. Install the two LEDs on the board and add code to light up one or the other based on the local range threshold and the distance that has been measured.
11. Add the code to sound the buzzer. Use some of your code from the ADC lab to play a tone for a short time (1 second or less). Make the tone play whenever the remote distance is below the remote range threshold. For testing purposes, you can make it play based on the local range rather than the remote range.
12. Install the 74LS125 tri-state buffer. Write code to enable the the buffer after the program starts. Check that you can program the Arduino with the serial connections in place. This means the tri-state buffer is doing what it is supposed to do.

13. Write test code that continually sends data out the serial interface and use the oscilloscope to examine the output signal from the D1 port of the Arduino (transmitted data). For 9600 baud, the width of each bit should be 1/9600 sec. or 104μsec.

14. Write code to send the distance value in millimeters out the serial interface in the format specified in Sec. 4.6 and check with the scope that it's being transmitted after each measurement. Check that the transmitted packet matches the specified protocol with the start and end characters present.

15. Write code to receive the distance in mm and display the distance in cm on the LCD. This also should be done without using any floating point routines.

16. Do a "loopback" test of your serial interface. Connect the D1 (TX) pin to the input pin on the 74LS125 so you are sending data to yourself. Make a measurement and see if the remote distance is the same as the one you displayed on the LCD as the local distance.

17. Make sure your program is using the received remote distance in the code that determines whether or not to play the buzzer. With a loopback in place, check that the buzzer plays whenever the measured range is less than the remote range threshold.

At this point you have all the individual components working so it's time to check that everything is working as specified. Perform the following checks on your rangefinder.

1. Try taking measurements with an object at different distances from the sensor. The distances reported on the LCD should be reasonably close to the correct distance.

2. Confirm that the "Adjust" button toggles between the local and remote thresholds. Each time the mode changes, the correct threshold number should be displayed on the LCD.

3. Check that the rotary encoder can be used to adjust the minimum distance between 1 and 400 cm for both the local and the remote range threshold.

4. After changing the one or both threshold distances, cycle the power on your Arduino and check that the modified threshold distance is displayed when it starts up again. This shows that your data has been stored in the EEPROM correctly and you were able to read it back.

5. Take multiple range measurements and confirm that the green and red LEDs operate properly based on the local range threshold that you have set

6. Use three wires to hook your rangefinder to another one in the class. Between the boards connect ground to ground, transmitted data to received data, and received data to transmitted data. Confirm that both units can send their distance data to the other for displaying.

7. Check that if the received distance from the remote unit is below the remote range threshold value then the buzzer sounds.

Ultrasonic Ranging Module HC - SR04

Product features:

Ultrasonic ranging module HC - SR04 provides 2cm - 400cm non-contact measurement function, the ranging accuracy can reach to 3mm. The modules includes ultrasonic transmitters, receiver and control circuit. The basic principle of work:
(1) Using IO trigger for at least 10us high level signal,
(2) The Module automatically sends eight 40 kHz and detect whether there is a pulse signal back.
(3) IF the signal back, through high level , time of high output IO duration is the time from sending ultrasonic to returning.
Test distance = (high level time×velocity of sound (340M/S) / 2,

Wire connecting direct as following:

- 5V Supply
- Trigger Pulse Input
- Echo Pulse Output
- 0V Ground

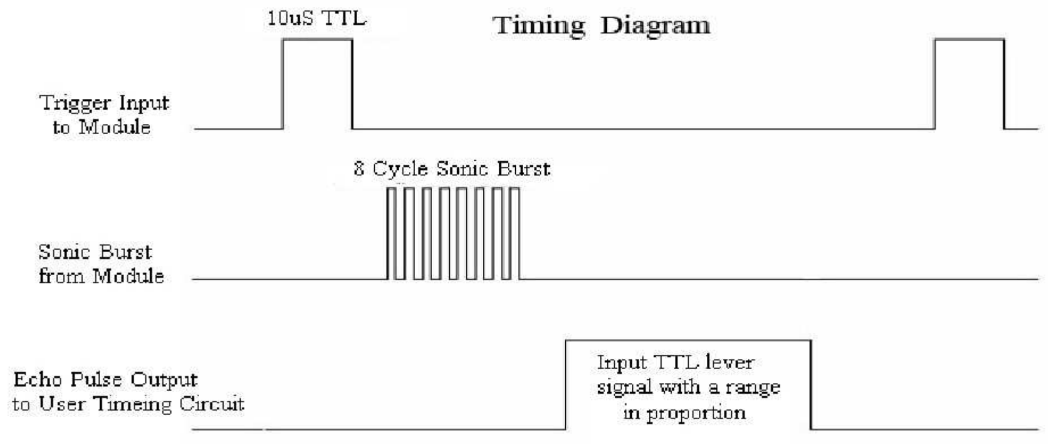
Electric Parameter

Working Voltage	**DC 5 V**
Working Current	**15mA**
Working Frequency	**40Hz**
Max Range	**4m**
Min Range	**2cm**
MeasuringAngle	**15 degree**
Trigger Input Signal	**10uS TTL pulse**
Echo Output Signal	**Input TTL lever signal and the range in proportion**
Dimension	**45*20*15mm**

Vcc Trig Echo GND

Timing diagram

The Timing diagram is shown below. You only need to supply a short 10uS pulse to the trigger input to start the ranging, and then the module will send out an 8 cycle burst of ultrasound at 40 kHz and raise its echo. The Echo is a distance object that is pulse width and the range in proportion .You can calculate the range through the time interval between sending trigger signal and receiving echo signal. Formula: uS / 58 = centimeters or uS / 148 =inch; or: the range = high level time * velocity (340M/S) / 2; we suggest to use over 60ms measurement cycle, in order to prevent trigger signal to the echo signal.


10uS TTL
Timing Diagram
Trigger Input to Module
8 Cycle Sonic Burst
Sonic Burst from Module
Echo Pulse Output to User Timeing Circuit
Input TTL lever signal with a range in proportion